

# webserv: Construindo um servidor Web sem bloqueio em C++ 98 (A42projeto)



MannBell · [Seguir](#)

5 minutos de leitura · 14 de janeiro de 2024



Ouvir



Compartilhar



No vasto domínio dos protocolos da Internet, o Protocolo de Transferência de Hipertexto (HTTP) permanece como a espinha dorsal da comunicação da World Wide Web. Para compreender verdadeiramente as complexidades da comunicação na web, [42](#) apresentou-nos um projecto maravilhoso para embarcarmos, que desafia as nossas capacidades e compreensão como sempre... Sim, estamos a falar do projecto “Webserv”, que envolve a criação de um servidor HTTP personalizado em C++98, com particular ênfase no não-bloqueio comportamento.

## Visão Geral do Projeto:

— Deve ser um servidor sem bloqueio.

Construir um servidor web em C++ envolve vários desafios, e um problema comum está relacionado ao bloqueio de operações. No contexto dos servidores web, o bloqueio refere-se à situação em que um thread está aguardando a conclusão de uma operação antes de poder prosseguir. Isso pode levar a gargalos de desempenho, especialmente em cenários onde o servidor precisa lidar com **diversas conexões simultâneas** .

O problema de bloqueio mais significativo surge *ao lidar com operações de entrada/saída (E/S), como leitura ou gravação em soquetes* . Em um modelo de bloqueio tradicional, um thread aguardará até que os dados estejam disponíveis ou até que uma operação de gravação seja concluída. Durante esse período de espera, o thread fica essencialmente inativo e o servidor pode ter dificuldades para lidar com outras solicitações recebidas.

Para resolver isso, empregaremos E/S assíncrona; A E/S assíncrona permite que um servidor inicie uma operação de E/S e continue com outras tarefas enquanto aguarda a conclusão da operação. Dessa forma, um único thread pode gerenciar múltiplas conexões simultaneamente sem ser bloqueado.

A E/S assíncrona pode ser feita usando várias chamadas de sistema disponíveis, como `select` , `poll` , `epoll` (Linux)...

`poll` e `epoll` são alternativas mais simples/melhores `select` , mas `select` são mais portáteis (podem ser usados em diferentes plataformas), então vamos falar um pouco sobre isso:

`select` é uma chamada de sistema em sistemas operacionais do tipo Unix que facilita o monitoramento assíncrono de vários descritores de arquivo para eventos específicos, como leitura, gravação ou erros. Ele permite que um programa espere e manipule eventos com eficiência em um conjunto de descritores. Aqui está uma visão geral de como `select` funciona:

de bits que representam conjuntos de descritores de arquivos ( *para que possamos saber quais FDs estão prontos para leitura/gravação, bit 1 para prontos e 0 para não*).

## 2. Preenchendo o fd\_set:

Os descritores de arquivo de interesse são adicionados ao respectivo fd\_set usando macros como FD\_SET .

## 3. Configurando o Tempo Limite:

Podemos definir um valor de tempo limite, para determinar a duração máxima select que devemos esperar por um evento. Se o tempo limite expirar sem nenhum evento, select ele retorna, permitindo que o programa prossiga.

## 4. Chamando select :

O programa invoca a select função, fornecendo o "highest file descriptor value" + 1 ( *man select* ), o three fd\_sets e o timeout ; Podemos definir NULL como valor para eventos nos quais não temos interesse, ou para timeout caso desejemos select bloquear indefinidamente até que ocorra um evento.

## 5. Bloqueando ou Retornando:

select entra em estado de bloqueio, aguardando eventos nos descritores de arquivos monitorados ou até que o tempo limite expire. Após um evento ou tempo limite, select retorna o controle ao programa.

## 6. Verificando o fd\_set:

Após retornar, o programa examina o fd\_sets para identificar quais descritores de arquivos estão prontos para os eventos especificados, usando macros como FD\_ISSET .

## 7. Tratamento de eventos:

podemos responder a eventos executando as ações necessárias com base nos descritores de arquivo identificados. Por exemplo, podemos ler/escrever dados de/para um soquete ou tratar erros.

## 8. Loop ou saída:

Muitas vezes empregamos um loop para repetir o processo de monitoramento, permitindo o tratamento assíncrono contínuo de eventos em vários descritores de arquivo.

como afirma o manual:

**WARNING:** `select()` can monitor only file descriptors numbers that are less than **FD\_SETSIZE** (1024)—an unreasonably low limit for many modern applications—and this limitation will not change. All modern applications should instead use `poll(2)` or `epoll(7)`, which do not suffer this limitation.

<https://man7.org/linux/man-pages/man2/select.2.html>

## — Implementando o HTTP/1.1.

Depois de configurar o servidor para operações básicas de leitura e gravação, podemos agora analisar o que trocamos com os clientes (ou seja, mensagens).

Uma comunicação HTTP típica consiste em compreender as solicitações e gerar respostas apropriadas e, para isso, é crucial compreender a estrutura das mensagens HTTP.

Vou abordar levemente o HTTP/1.1 aqui, mas caso você esteja procurando mais profundidade, aconselho a ler as RFCs oficiais. É essencial focar nas versões mais recentes - *investi um tempo considerável lendo rfc2616, apenas para perceber que está obsoleto há muito tempo* ; no momento da redação deste artigo, os atuais são [rfc9110](#) e [rfc9112](#) .

As mensagens HTTP consistem em uma linha de solicitação ou resposta, cabeçalhos, uma linha vazia ( `CRLF` ou `\r\n` ) e um corpo de mensagem opcional.

Aqui está uma ilustração usando [o Formulário Backus – Naur Aumentado](#) :

```
Mensagem HTTP = linha inicial CRLF
                  *(linha de campo CRLF)
```

## 1. Linha de solicitação:

linha de solicitação = método SP destino da solicitação SP versão HTTP

A linha de solicitação em uma mensagem de solicitação HTTP contém o método, o URI e a versão HTTP. Analisar esta linha é o primeiro passo para entender a solicitação do cliente.

Por exemplo:

OBTER /caminho/ para /recurso HTTP/ 1.1

- No exemplo acima, “GET” é o método, “/path/to/resource” é o URI e “HTTP/1.1” é a versão.

## 2. Linha de status/linha de resposta

linha de status = versão HTTP SP código de status SP [frase de motivo]

Da mesma forma, em uma mensagem de resposta HTTP, a linha de status/resposta contém a versão HTTP, o código de status e uma frase de motivo. Extrair essas informações é crucial para gerar respostas significativas.

Por exemplo:

HTTP/1.1 200 OK

### 3. Cabeçalhos.

Tanto as solicitações quanto as respostas podem incluir cabeçalhos, fornecendo informações adicionais sobre a mensagem. A análise de cabeçalhos envolve a extração de pares de valores-chave para compreender vários aspectos, como tipo de conteúdo, comprimento do conteúdo e muito mais.

Por exemplo:

```
Host: example.com  
Tipo de conteúdo: text/html  
Comprimento do conteúdo: 256
```

A análise desses cabeçalhos permite que o servidor interprete o conteúdo de maneira adequada.

### 4. Corpo da mensagem:

O corpo da mensagem, embora opcional, pode conter dados relevantes para a solicitação ou resposta. A análise do corpo da mensagem depende de fatores como tipo e comprimento do conteúdo, codificação de transferência, etc.

Depois de terminar a análise das mensagens HTTP, as próximas etapas envolvem a implementação de lógica para lidar com diferentes métodos HTTP, processar solicitações e gerar respostas apropriadas.

Aqui está um exemplo de fluxograma que criei, para ter uma visão geral do processo de resposta:

